# Trax & BGU Deep Learning Student Project - Nir Sela & Noam Cohen

## Summary

### Abstract

The goal of this project is to estimate the effect of a new approach for configuring Convolutional Neural Networks, on image classification in a dataset used by ‘Trax Image Recognition’.

This project was conducted under the guidance of Dr. Oren Freifeld from BGU, and Aviv Eisenschtat from Trax.

This new approach was presented in 2016 by Trang Pham, Truyen Tran, Dinh Phung and Svetha Venkatesh from Deakin University, Australia, in their paper “Column Networks for Collective Classiﬁcation”.

In our project, we took the ideas presented and applied them on a dataset provided by Trax.

This document describes our implementation, research, results and conclusions, alongside the difficulties we encountered along the way, and how we’ve managed to overcome them.

Source code, logs, and result files are attached.

### The Concept

in supervised learning for object classification, we take a set of examples, alongside labels assigned to them, and train a classification model to classify these examples to their correct labels, and be able to classify examples it has not encountered yet in the future.

But, classic CNN’s for image classification handle one example at a time, without trying to exploit certain relations between examples that might help with correct classification.

The idea presented in the paper we’ve implemented wishes to use relations between examples, by using a concept they call a Column Network (CLN). In the concept, the network is built as parallel mini-columns, each is a network by itself, joint together to a single network.

Communications between mini-columns are enabled through short-range horizontal connections. In CLN, each mini-column is a feedforward net that takes an input vector – which plays the role of a receptive ﬁeld – and produces an output class. Each mini-column net not only learns from its own data but also exchanges features with neighbor mini-columns along the pathway from the input to output.

The relations between the columns are directly based on the relations defined on the input set of examples.

Each example is marked with a few sets of relations, in each set are all the other examples that stands in this relation with the example.

So, instead of feeding the network one example at a time, when given the example, and it can exploit other examples and information about them to classify correctly.

To better understand the theoretical details behind this idea, the paper is attached.

### Adapting to The Trax Dataset

Our dataset consists of images of shelves ins stores, mainly containing images of beverages.

The idea was to test the effect of the spatial location of a bottle, alongside the other ones around it, to improve the fine-tuned classification that Trax requires.

Each image was split to sub images of the bottles it contains, and those we assigned spatial relations of IsOnRight\ IsOnLeft with one another.

The hope was that information of a products surrounding will help in its classification.

This information was given as input to a network built using the architecture described above.

### Conclusions

The performance of the network we built were compared to those of a classic CNN, that only accepts labeled examples as input and classifies them.

Unfortunately, the comparison between the two did not show a significant difference in their performance, and we were not able to achieve a major improvement in classification on our dataset.

The full work process and exact results, alongside our attempts to improve them, are described ahead.

## Preparing the Dataset

### Structure

The source data we’ve received from Trax consisted of 16487 images of shelves in stores, alongside a csv file describing all the products on these shelves, recognized by Trax. In these images, there is a total of 290763 products, distributed amongst 823 labels.

Each product was characterized by:

* Id – product id.
* Brand label – label of the brand family this product belongs to. 141 different brands.
* Mask – coordinates for location of a bounding box for the product in the image it belongs to.
* Patch URL – an identifier for the product.
* Probe ID – the image this product belongs to.
* Product label – the true label given to this product by Trax. 823 different labels.

### Image Cropping and Resizing

The first step was separating the images of the shelves to distinct images of products.

Using the images, csv files and the coordinates of each product in its image, we created this image.

Since the bounding boxes define images of different sizes, but Neural Networks expect to have image of the size as their input, we have had to resize images to be of a constant size.

In the end, to improve performance and to align with the size convention used in Trax, we have resized images to 100x300 pixels, using padding in case images need to be enlarged.

we have tried padding images using zero padding or random Gaussian noise. There was no significant difference between the two, so in the end we’ve simply used zero padding.

### Defining Relations

We have decided to define two relations between different products:

1. IsOnRight(x, y): products x and y are on the same shelve, y is THE product to the right of x.
2. IsOnLeft(x, y): products x and y are on the same shelve, y is THE product to the left of x.

Both the relations are NOT transitive. For example, not all products on the same shelve as x on its right stand in the relation IsOnRight with it, but only the closest one adjacent to it.

To calculate these relations, we have used a variation of the DbScan algorithm.

DBSCAN (Density-based spatial clustering of applications with noise) is an algorithm that divides a set of elements to foreign subsets (clusters) based on the proximity between the elements, which defined with a distance function on every pair of elements.  
If two elements are adjacent (i.e. the distance between them is small enough), both of those elements, and any element that is adjacent in transitive manner, will be in the same cluster.  
In our case, each cluster is a group of bottles standing on the same shelf. So the measured distance is the gap between the lower y-value of each element's mask.  
Now, each cluster (shelf) is sorted by the x-values of the elements mask, and the IsOnRight/IsOnLeft relations can be calculate.

### Creating Datasets

From the data provided, we have created 2 datasets. One consists of almost all the images and products (we only removed the last items for convenience), and the other is a filtered dataset that contains only products that have sufficient examples from the same label for the network to be able to learn how to classify them.

We decided to try and filter out data since when we first ran the network, using both CLN\CNN architectures, the classification accuracy was very poor.

It came to mind that there may many labels in the dataset that don’t have enough examples to train on, so have tried to evaluate performance only on those that have.

The filtering criteria was to classify only on labels that have at least a 1000 examples.

Filtering reduced the total amount of products to ~200k, roughly 67% of the original size.

Next, we split every dataset into three subsets: training, validation and test sets.

Set sizes are:

|  |  |
| --- | --- |
|  | #Products |
| Filtered-Train | 127601 |
| Filtered-Validation | 17220 |
| Filtered-Test | 10646 |
| Full-Train | 224506 |
| Full-Validation | 31292 |
| Full-Test | 29452 |

Eventually, every dataset consists of the following lists:

* Paths: a list of full paths to the location of the resized-cropped image of the product. We don’t provide actual features(pixel) for scalability. They will be read from disk during execution.
* Labels: a list of labels for products.
* Relation List: a list of relations for every product. Each member of a list describes the relations IsOnRight and IsOnLeft.
* Batches: define the image the product originated from.

## Running the Network

As mentioned before, we have written our code to support a few variations of Neural Networks, to compare their performance and test if using CLN relations to improve classification is suitable for the Trax dataset.

However, all configurations follow the same basic concept that splits the network into two parts:

* Converting images to feature vectors: this is the input CLN’s expects, so we used different methods to achieve this: simple flatting, Convolutions, Global Pooling etc.
* Running through CLN, if enabled.

### Configuration Parameters

In order to best explain the different variation, the following is an overview of the configuration parameters our code is given, and brief explanation of this parameter on the results.

If a parameter is assigned a default value, it means we used this value through all our executions, and did not cross validate its effect on the results.

|  |  |  |
| --- | --- | --- |
| Parameter name | Description | Default value |
| Dataset | The name to the dataset to learn. Can be either the full dataset, or the filtered one.  The logic behind filtering has been explained. |  |
| Model type | The type on NN to build: normal CNN, or the new CLN. The main goal was to compare between these two architectures to test if CLN’s provide a benefit in classification. |  |
| Flat Method | A method to convert an image to a feature vector. Can be either a simple flatting, or running through a CNN consisted of Convolutions, pooling and Activations. | CNN |
| Pooling | Use Global Max Average Pooling layers or not. This was intended to better fit images to CLN’s |  |
| Batch Type | Constant: feed the network data in constant batches.  Probe: feed the network in batches of all products in the same image, or a few images at once. In practice, we did not use this method since it has shown to be extremely slow.  Random: randomize order of products at the beginning of every epoch and use a constant batch size. |  |
| Batch size | Using Probe Batch Type, this parameter indicates how many images to feed the network every time. | 5 |
| Constant Batch Size | Using Constant\Random Batch Type, this is the batch size used. | 128 |
| Number of Epochs | Number of epochs to train the network on. | 100 |
| Optimizer | The optimizer used in Model. Can be wither RMS\Adam. | RMS |
| Learning Rate | The learning rate parameter for the optimizer. | 0.0001 |
| Seed | A seed for random methods. | 1234 |
| Dropout | Use dropout or not. | True |
| Number of layers | Number of layers in each Column. See paper for further info. | 10 |
| Dim | Number on nodes in each layer of each Column. See paper for further info. | 400 |
| Shared | Share Parameters between Columns or not. See paper for further info. | True |
| N mean | Number of relation types. See paper for further info. | 2 |

### Selected Configurations

Below are the configurations we choose to test. As mentioned before, all other parameters are set to their default value.

|  |  |  |  |
| --- | --- | --- | --- |
| Dataset | Batch Type | Pooling | Model Type |
| Filtered | Constant | True | HCNN |
| Filtered | Constant | True | CNN |
| Filtered | Constant | FALSE | HCNN |
| Filtered | Constant | FALSE | CNN |
| Filtered | Random | True | CNN |
| Full | Constant | True | HCNN |
| Full | Constant | True | CNN |

### Infrastructure

Our code is written in python, using the Keras Deep Learning Library with TensorFlow backend.

Our executions were made using a GPU server at BGU labs.

Server details:

TBD-NOAM.

### Evaluating Parameters

Throughout executions, we have tested the following standard learning parameters:

* Loss.
* Accuracy.
* F1.
* Recall.
* Precision.

### Program Flow

Running the program roughly follows this flow:

* Process input parameters.
* Load dataset from disk.
* Build a Keras NN model, using the requested architecture: CNN\CLN.
* Log information about model.
* Create sub samples: training, validation and testing using information provided.
* Create Python Generators on these sub samples, using the requested batch type and batch size.
* Define callbacks to be used every time an epoch ends, to log the performance of the network on the validation\testing sub samples.
* Train the model on the training set using ‘fit generator’ technique: this generates batches of examples for the network using a python generator. Our generator load data from disk, using the requested batch type.
* Logs all results.

## Results

This section will describe the results of every execution we have made.

See Results directory for raw files.

### Comparison

The table below compares the Accuracy Evaluation Parameter of the last epoch and Runtime of all executions.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Execution ID | Dataset | Batch Type | Pooling | Model Type | Exec? | Run Time | Test Accuracy | Validation Accuracy | Test Accuracy |
| 1 | Filtered | Constant | True | HCNN | 1 | 16h | 78% | 77% | 74% |
| 2 | Filtered | Constant | True | CNN | 1 | 11h | 78% | 76% | 72% |
| 3 | Filtered | Constant | FALSE | HCNN | 1 | 16h | 73% | 76% | 79% |
| 4 | Filtered | Constant | FALSE | CNN | 1 | 11.5h | 73% | 79% | 78% |
| 5 | Filtered | Random | True | CNN | 1 | 11h | 74% | 75% | 76% |
| 6 | Full | Constant | True | HCNN | 0.5 | 29h | 54% | 52% | 50% |
| 7 | Full | Constant | True | CNN | 0.5 | 21h | 60% | 57% | 57% |

The following charts describe the progress of each evaluation parameter, in every execution.

### Evaluation Parameters Progress

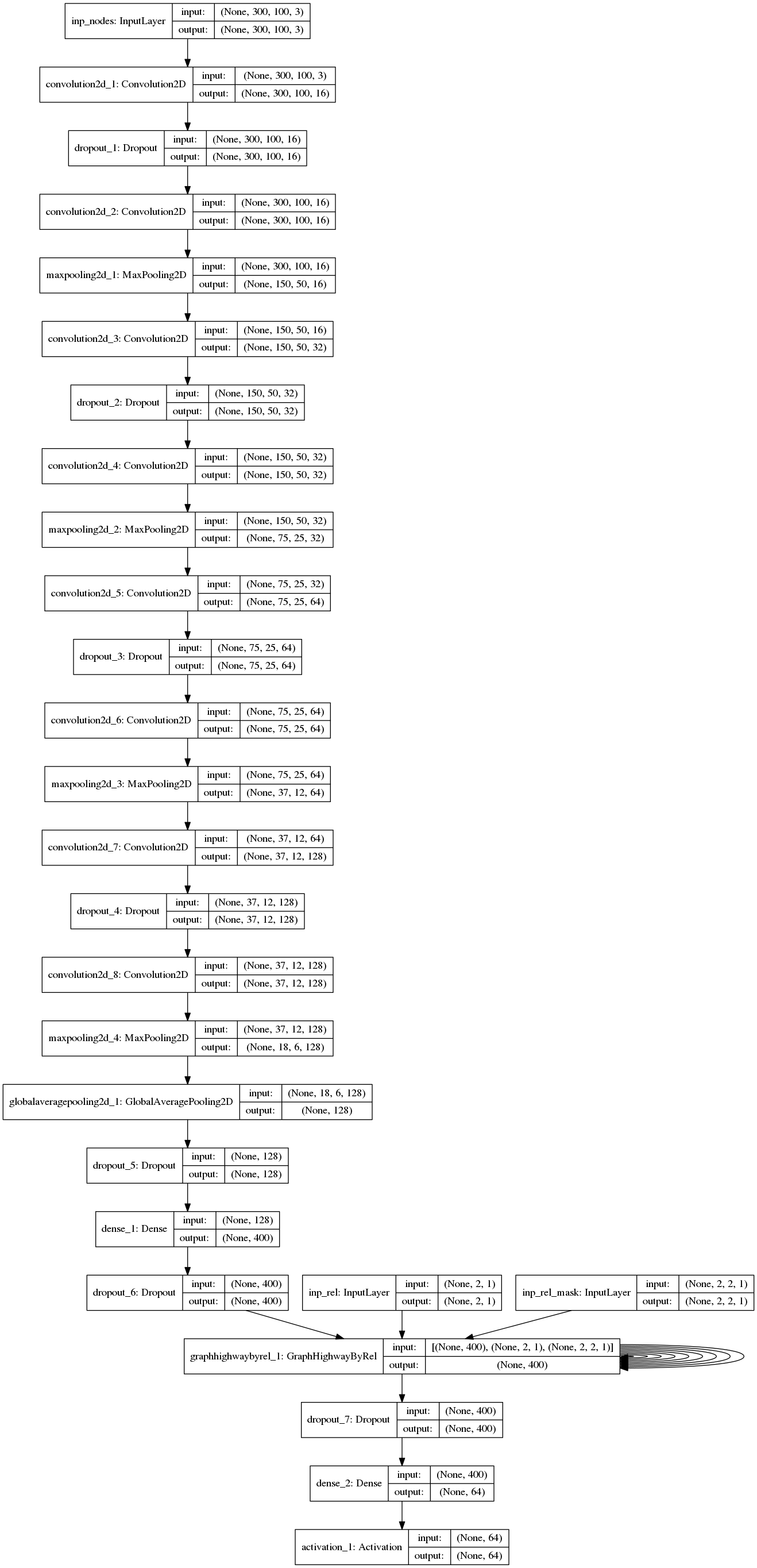
### Observations

In all executions, it can be observed that:

* All charts appear to converge to a certain value.
* All charts demonstrate an improvement over time.
* It appears that in most evaluation parameters, Execution 5 was the best one. It had the fastest convergence. Even so, it appears to be the most unstable of them all, in terms of progress.
* All evaluation parameters on filtered dataset are better than those of
* There is no significant difference between validation/test set (seen in raw logs).

### Execution 1

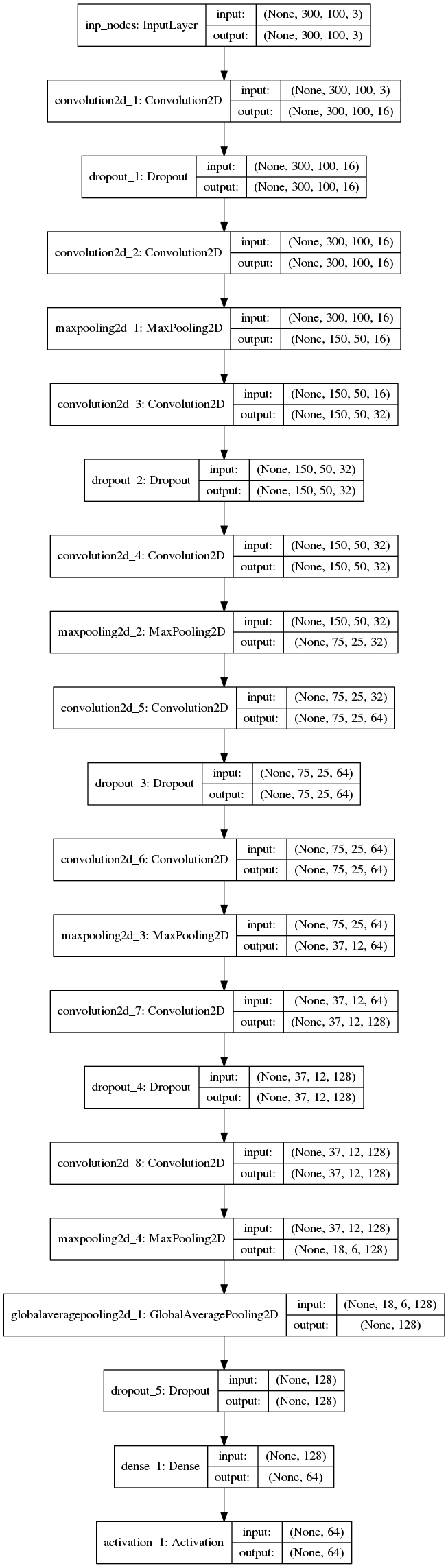
#### Model Structure



#### Evaluation Parameters progress

### Execution 2

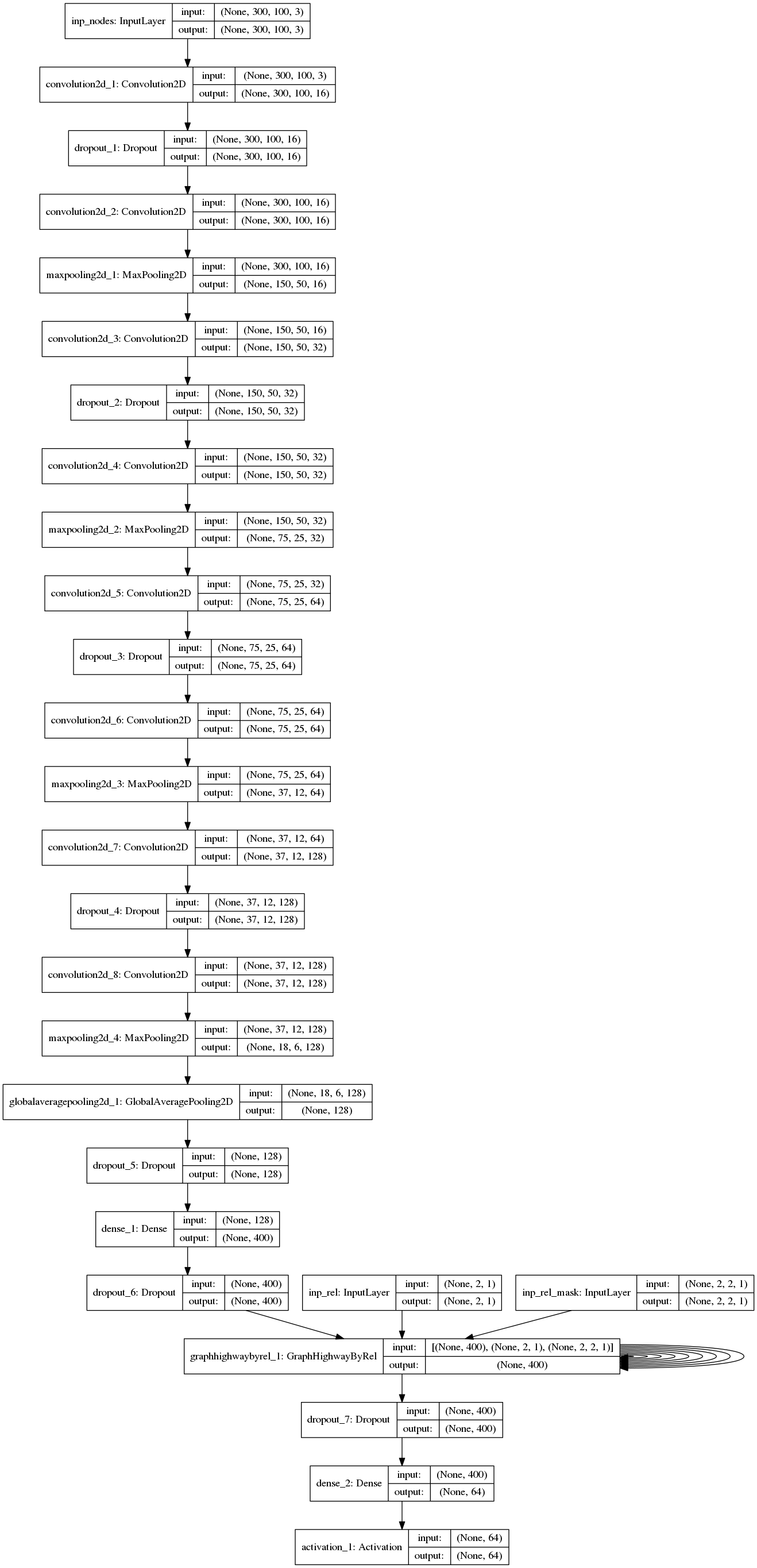
#### Model Structure



#### Evaluation Parameters progress

### Execution 3

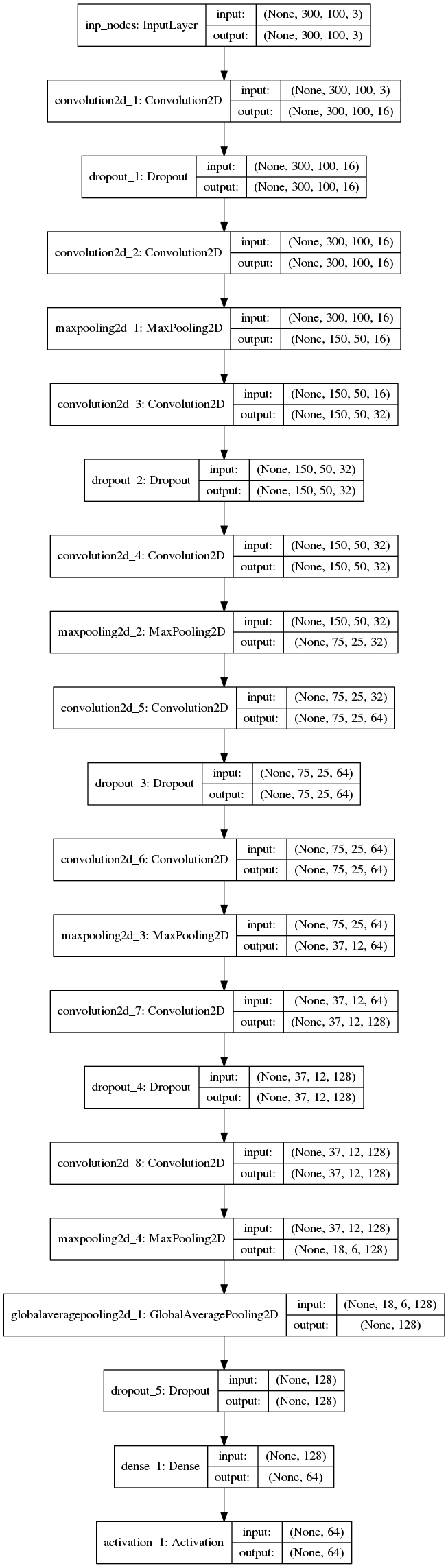
#### Model Structure



#### Evaluation Parameters Progress

### Execution 4

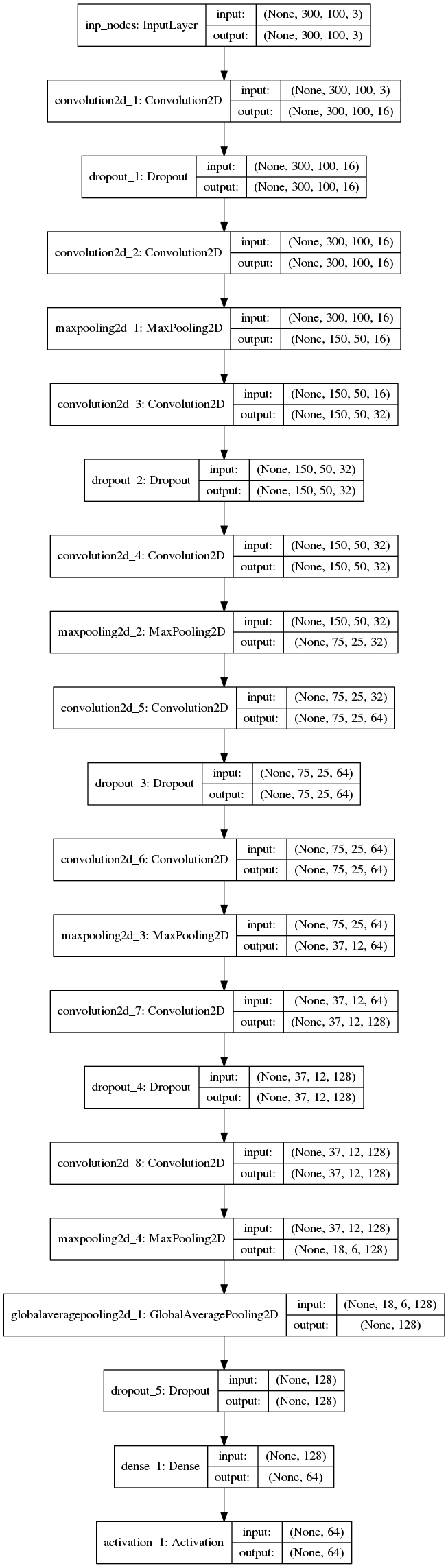
#### Model Structure



#### Evaluation Parameters Progress

### Execution 5

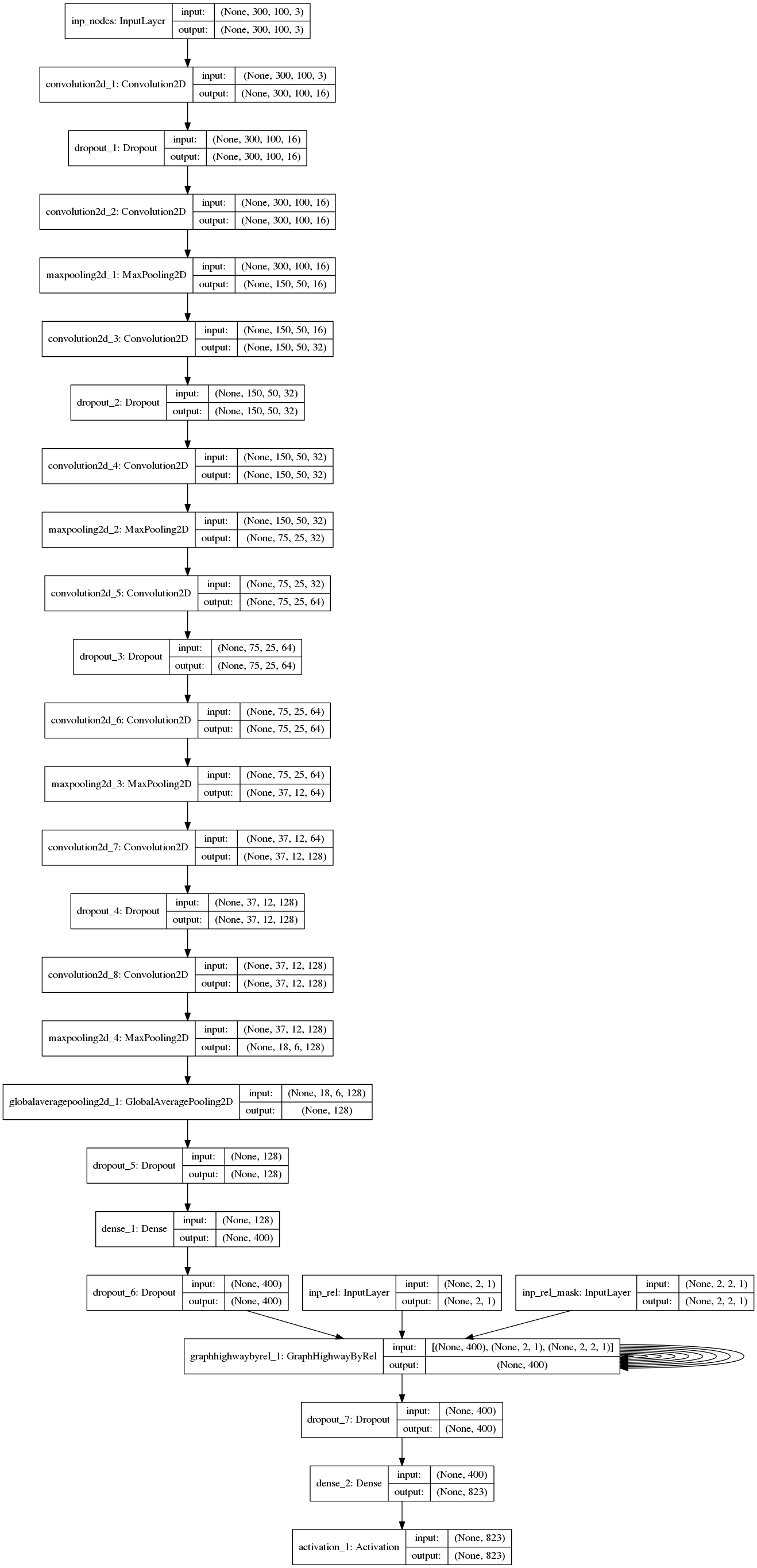
#### Model Structure



#### Evaluation Parameters Progress

### Execution 6

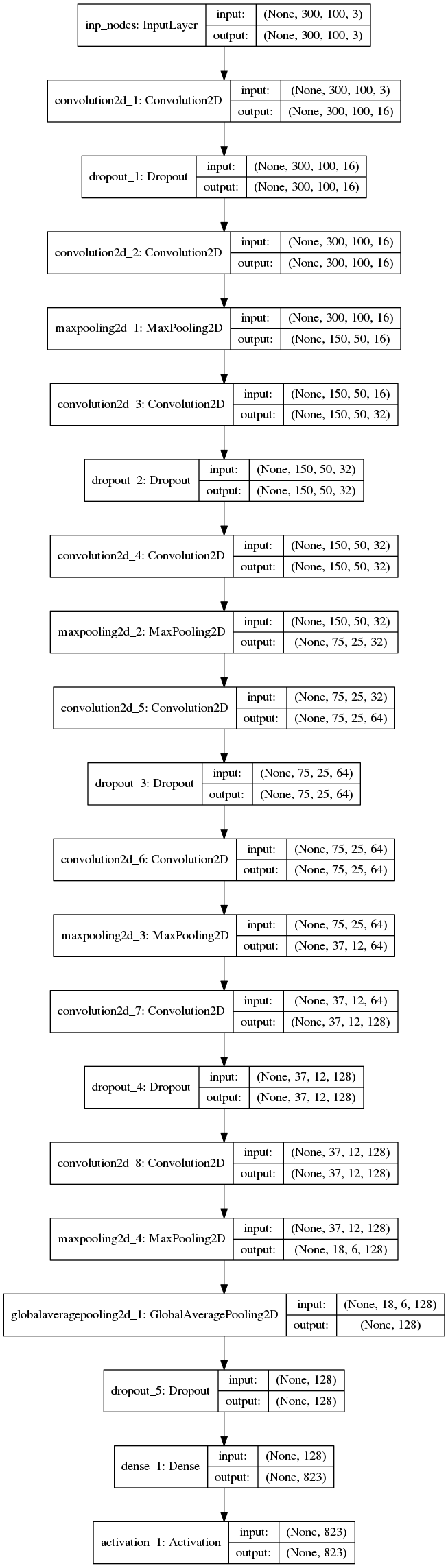
#### Model Structure



#### Evaluation Parameters Progress

### Execution 7

#### Model Structure



#### Evaluation Parameters Progress

## Conclusions

The results above lead to the following conclusions:

1. The main conclusion from this project is that using the CLN architecture did not prove to be beneficial on Trax Data set in its current design. Further research may be able to achieve a breakthrough and configure it properly, but this was not the case. In addition, we note that:
2. The best result we achieved is Execution 5: using CNN, randomized batches, with max pooling.
3. Evaluation parameters on the full dataset are not as good as those on the filtered dataset. There are techniques of artificially augmenting the dataset, so that less common labels will have more examples, but this appears to be irrelevant since there is no significant difference between CLN\CNN, so this will not prove that CLN can assist in classification task.
4. Running the CLN architecture is slower.
5. The convergence of the parameters make it unlikely that running for more epochs will make a difference performance.